

Regelgesteuerte Prozesse und Rule Engines

von

Robert Switzer

1 Einleitung

Manche Geschäftsprozesse können durch eine relativ kleine Menge von Regeln beschrieben werden. Es hat Vorteile, wenn man diese Regeln nicht in Programmcode gießt (hardwired), sondern sie in einem externen Dokument formuliert, das von der Anwendung gelesen und interpretiert wird. Dann können die Regeln leichter modifiziert werden. Erfahrungsgemäß werden Geschäftsregeln relativ häufig geändert.

Beispiel. Die Geschäftsregeln für ein Zoogeschäft könnten Regeln beinhalten, die so aussehen:

1. Wenn der Kunde mindestens einen Goldfisch in seinem Warenkorb hat und noch kein Fischfutter, leg ihm eine kostenlose Probe unseres Fischfutters in den Korb.
2. Wenn der Kunde mindestens fünf Goldfische in seinem Warenkorb hat und noch kein Aquarium, frag ihn, ob er ein Aquarium für seine Fische kaufen möchte. Wenn er mit “Ja” antwortet, leg ihm ein Aquarium in den Korb.
3. Wenn der Gesamtwert der Ware im Korb mindestens 10,00 Euro und weniger als 20,00 Euro beträgt und der diesem Kunden bereits gewährte Rabatt weniger als 5% ist, gewähr ihm einen Rabatt von 5%.
4. Wenn der Gesamtwert der Ware im Korb mindestens 20,00 Euro beträgt und der diesem Kunden bereits gewährte Rabatt weniger als 10% ist, gewähr ihm einen Rabatt von 10%.

Damit diese Idee funktionieren kann, muß es irgendwo einen Agenten geben, der die betreffenden Regeln interpretieren und die in ihnen spezifizierten Aktionen ausführen kann. Diesen Agenten nennt man allgemein *rule engine*.

Da diese Idee so bestechend ist, sind eine Vielzahl von rule engines entwickelt worden – teils kommerziell, teils im Opensourcebereich. Das führte zu einer Vielzahl von APIs, die Anwender beherrschen mußten, um solche rule engines in ihren Anwendungen einzusetzen. Dieses Wirrwarr führte wiederum zum Ruf nach einem neuen Java Specification Request (JSR). So entstand der JSR 94 (s. [jsr94]), der ein API für rule engines festlegt.

Ähnlich wie bei dem Java Message Service unterscheidet man hier zwischen *Client* und *Provider*. Der Client ist die jeweilige Anwendung, die den rule engine einsetzen soll; der Provider ist die Implementation des rule engine. Die einen Implementatoren implementieren den rule engine als Teil der Anwendung – d. h. er läuft auf derselben JVM wie

die Anwendung – und andere implementieren ihn als separaten Prozeß, der evtl. über das Netz erreicht wird.

[jsr94] äußert sich ausdrücklich *nicht* zu der Gestalt des Dokuments, in dem die Regeln festgehalten werden. Die Hersteller haben hier freie Hand. Man darf wohl davon ausgehen, daß diese Dokumente in fast allen Fällen XML-Dokumente sein werden.

2 Einige Definitionen

Hier will ich einige grundlegenden Definitionen aus [jsr94] bringen, da sie im folgenden eine wichtige Rolle spielen werden.

Rule Engine Dieser Begriff steht naturgemäß im Mittelpunkt von [jsr94]. [jsr94] beschreibt einen rule engine als raffinierten Interpreter für *if/then*-Aussagen.

Der *if*-Teil beinhaltet Bedingungen wie etwa

```
cart.totalAmount > 20.00
```

und der *then*-Teil solche Aktionen wie etwa

```
giveDiscount(0.10);
```

Die Eingabe zum rule engine besteht aus einer Menge von Regeln und einigen Datenobjekten. Die Ausgabe wird natürlich von der Eingabe bestimmt und kann bestehen aus den Eingabeobjekten (evtl. mit Modifikationen) neuen Datenobjekten und Nebeneffekten wie etwa

```
sendMail("Thank you for shopping with us")
```

[jsr94] weist darauf hin, daß es große Unterschiede zwischen Provider geben kann, nennt jedoch folgende Gemeinsamkeiten:

- die Förderung von deklarativer Programmierung, indem Geschäfts- bzw. Anwendungslogik externalisiert wird.
- alle beinhalten ein dokumentiertes Format für das Dokument, welches die Regeln formuliert – oder aber Werkzeuge extern zum rule engine, mit denen Regeln formuliert werden können.
- sie wirken auf Eingabeobjekte, um Ausgabeobjekt zu generieren. Eingabeobjekte werden häufig *Fakten* genannt und repräsentieren den Zustand des Anwendungsbereichs. Ausgabeobjekte werden häufig *Folgerungen* oder *Konsequenzen* genannt und deren Bedeutung im Anwendungsbereich wird durch die Anwendung selbst geliefert.

- in einigen Fällen führt der rule engine Aktionen aus, die den Zustand des Anwendungsbereichs, die Eingabeobjekte, den Interpretationsprozeß, die Regeln oder den rule engine selbst beeinflussen.
- oder der rule engine kann lediglich Ausgabeobjekte generieren und es der Anwendung überlassen, diese zu interpretieren bzw. Aktionen auszuführen, die diese implizieren.

Rule Eine Regel (engl. rule) besteht gewöhnlich aus zwei Teilen: einer Bedingung und einer Aktion. wenn die Bedingung erfüllt ist, wird die Aktion ausgeführt. [jsr94] spezifiziert die genauere Beschaffenheit einer Regel nicht, da es erhebliche Unterschiede von Provider zu Provider gibt. [jsr94] legt lediglich fest, daß jede Regel primitive Metadaten bestehend aus Namen plus Beschreibung zur Verfügung stellen soll.

Rule Execution Set Eine Menge von Regeln wird als *rule execution set* bezeichnet. Auch hier spezifiziert [jsr94] lediglich, daß eine rule execution set primitive Metadaten bestehend aus Namen und Beschreibung zur Verfügung stellen soll.

Rule Session Eine rule session ist eine Laufzeitverbindung zwischen Client und Provider. Mit einer rule session kann nur *eine* rule execution set assoziiert werden. Da eine rule session u. U. erhebliche Ressourcen im rule engine binden könnte, sollte eine rule session stets freigegeben werden, wenn sie nicht mehr benötigt wird.

Stateful Rule Session Eine zustandsbehaftete (stateful) rule session gestattet dem Client eine länger dauernde Interaktion mit dem rule engine durchzuführen. Eingabeobjekte können sukzessive zur session hinzugefügt und Ausgabeobjekte können wiederholt abgefragt werden.

Stateless Rule Session Es handelt sich hier um ein performantes und einfaches API, das eine rule execution set mit einer Liste von Eingabeobjekten ausführen kann. Die Methoden dieses API sind idempotent.

3 Die Architektur

Ähnlich wie beim JMS trennt [jsr94] die Funktionalität, die jedem Client zur Verfügung steht, von der Funktionalität, die dem Administrator zur Verfügung gestellt wird. Es gibt dementsprechend die beiden Packages

```
javax.rules
```

und

```
javax.rules.admin
```

Die Vorstellung hierbei ist, daß eine Anwendung eine rule execution set ausführen wird, die ein Administrator zuvor in den rule engine geladen und registriert hat. So könnte man festlegen, daß einige Benutzer Regeln ausführen dürfen, diese jedoch nicht administrieren.

4 Typische Anwendungsszenarien

4.1 Lokaler Rule Engine

Wenn der rule engine lokal zur Anwendung laufen soll und nicht etwa in einem Anwendungsserver oder gar als Daemon, könnte der Client wie folgt aussehen:

```
import java.io.InputStream;
import java.util.LinkedList;
import java.util.List;

import javax.rules.*;
import javax.rules.admin.*;

public class Mixed
{
    private static final String PROVIDER_CLASSNAME =
        "org.drools.jsr94.rules.RuleServiceProviderImpl";
    private static final String DROOLS_URI = "http://drools.org/";
    private static final String RULE_URI = "rules.xml";
    public static void main(String[] args)
    {
        try
        {
            // load service provider.
            Class.forName(PROVIDER_CLASSNAME);
            RuleServiceProvider serviceProvider =
                RuleServiceProviderManager.getRuleServiceProvider(
                    DROOLS_URI);

            // In this section we are administrator.

            // get the rule administrator
            RuleAdministrator adm =
                serviceProvider.getRuleAdministrator();

            // get an input stream to read the rule document
            InputStream rules =
                Mixed.class.getResourceAsStream(RULE_URI);

            // parse the ruleset from the XML document

            LocalRuleExecutionSetProvider lresp =
                adm.getLocalRuleExecutionSetProvider(null);
            RuleExecutionSet res =
                lresp.createRuleExecutionSet(rules, null);
        }
    }
}
```

```

rules.close();

// register the rule execution set
adm.registerRuleExecutionSet(RULE_URI, res, null);

// And from here on we are client.

// create a stateless rule session
RuleRuntime rt = serviceProvider.getRuleRuntime();
StatelessRuleSession session = (StatelessRuleSession)
    rt.createRuleSession(
        RULE_URI,
        null,
        RuleRuntime.STATELESS_SESSION_TYPE);

// provide some input objects
List input = new LinkedList();
// add some input objects to input
// ...

// execute the rules
List output = session.executeRules(input);

// and inspect the output objects
System.out.println("Output from rule set execution");
for (Object obj : output)
{
    System.out.println(obj.toString());
}
} catch (Exception e)
{
    System.err.println(e);
}
}
}

```

Kommentar:

1. Dieser code spielt zwei Rollen: Client und Administrator. Da der rule engine im Prozeß des Client läuft, hätte ein externer Administrator keine Möglichkeit, den rule engine zu erreichen, den unser Client benutzt, um dort eine rule execution set zu registrieren. Darum wird hier Administratorcode mit Clientcode vermischt. Deshalb habe ich diese Klasse *Mixed* genannt.
2. Die Art, wie hier auf den service provider zugegriffen wird, erinnert stark an die Art, wie unter JDBC 1.0 eine Verbindung zum Datenbankprovider geholt wurde. Seit

JDBC 2.0 benutzt man das JNDI, um direkt eine Datenquelle (Typ `DataSource`) zu erhalten. Daran wird der Code im nächsten Abschnitt erinnern.

4.2 Ein Rule Engine mit JNDI-Vermittlung

Nun wollen wir sehen, wie Clientcode aussehen könnte, wenn der rule engine in einem Anwendungsserver oder als Daemon läuft:

```
import java.util.LinkedList;
import java.util.List;

import javax.naming.*;
import javax.rules.RuleRuntime;
import javax.rules.StatelessRuleSession;

public class RuleClient
{
    private static final String RULE_URI = "rules.xml";

    public static void main(String[] args)
    {
        try
        {
            InitialContext initial = new InitialContext();
            RuleRuntime rt = (RuleRuntime) initial.lookup("myRules");
            StatelessRuleSession session = (StatelessRuleSession)
            rt.createRuleSession(
                RULE_URI,
                null,
                RuleRuntime.STATELESS_SESSION_TYPE);

            // provide some input objects
            List input = new LinkedList();
            // add some input objects to input
            // ...

            // execute the rules
            List output = session.executeRules(input);

            // and inspect the output objects
            System.out.println("Output from rule set execution");
            for (Object obj : output)
            {
                System.out.println(obj.toString());
            }
        }
    }
}
```

```

        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}

```

Kommentar:

1. Hier machen wir die übliche Geschichte mit `InitialContext`; was wir unmittelbar bekommen, ist ein Objekt vom Typ `RuleRuntime`. Die Fummelei mit `ServiceProviderManager` entfällt ganz.
2. Dies funktioniert natürlich nur dann, wenn der Administrator ein entsprechendes rule execution set im rule engine installiert sowie einen Verweis darauf in den JNDI-Baum gehängt hat.

5 Der Rule Engine drools

Ich erwähnte schon, daß es diverse rule engines gibt. Einer aus dem Opensourcebereich hat den etwas seltsamen Namen `drools`. Ich will gleich einige einigermaßen realistische Beispielanwendungen zeigen, die mit `drools` implementiert wurden. Sie sind von den Beispielen, die mit `drools` mitgeliefert werden, abgekupfert und leicht nostrifiziert worden.

5.1 Die Drools Rule Language (DRL)

Ich sagte, [jsr94] äußert sich überhaupt nicht über die Gesalt der Dokumente, in denen Regeln formuliert werden; den Implementatoren bekommen hier vollkommen freie Hand. Darum müssen wir als erstes lernen, wie man unter `drools` Regeln formuliert. Hierzu sieht `drools` die Drools Rule Language (DRL) vor. Sie wird selbstverständlich auf der Basis der Metasprache XML konstruiert.

Die Grundstruktur eines DRL-Dokuments ist folgende:

- das Wurzelement heißt `rule-set` und muß ein Attribut `name` besitzen kann aber auch ein Attribut `description` haben.
- das Wurzelement kann beliebig viele `rule`-Subelemente besitzen, die wiederum ebenfalls ein Attribut `name` besitzen müssen; der Wert des `name`-Attributs muß im ganzen Dokument eindeutig sein.
- ein `rule`-Element kann beliebig viele `parameter`-Subelemente besitzen, wobei ein `parameter`-Element ein Attribut `identifizier` haben muß. Der Inhalt eines `parameter`-Elements ist in der Regel ein `class`-Subelement, dessen Inhalt der qualifizierte Klassenname des Parametertyps angibt.

- ein `rule`-Element kann beliebig viele `condition`-Subelemente haben und muß genau ein `consequence`-Subelement besitzen.

Die Semantik ist folgende:

- die Werte der Parameter können in den `condition`-Elementen sowie in dem `consequence`-Element vorkommen.
- der Inhalt eines `condition`-Elements sollte ein Boole'scher Ausdruck sein.
- der Inhalt eines `consequence`-Elements sollte eine ausführbare Instruktion sein.
- `drools` "führt" eine Regel aus, indem es alle `condition`-Elemente auswertet. Ergeben alle den Wert `true`, wird das `consequence`-Element ausgeführt. Ansonsten bleibt die Regel ohne Wirkung.

Der intelligente Leser müßte jetzt Fragen "In welcher Sprache werden Bedingungen und Instruktionen formuliert?" `drools` unterstützt natürlich Java und außerdem einige Skriptsprachen wie z. B. Python und Groovy.

Für jede dieser Sprachen gibt es in DRL einen eigenen Namensraum. Setzt man Java ein, sieht ein `condition`-Element so aus

```
<java:condition>...</java:condition>
```

Dabei ist der Inhalt des Elements natürlich ein Java-Ausdruck mit einem Boole'schen Wert. Und unter Java sieht ein `consequence`-Element so aus

```
<java:consequence>
...
</java:consequence>
```

wobei der Inhalt ein Block von Javacode ist.

5.2 Der Arbeitsspeicher von drools

`drools` bezeichnet die Gesamtheit aller Fakten (Eingabeobjekte) als *Wissen* (engl. knowledge). Dieses Wissen wird in dem *working memory* gespeichert.

Die `consequences` können dieses Wissen mit Hilfe des Interface

```
org.drools.spi.KnowledgeHelper
```

manipulieren, das folgende Methoden besitzt:

```
// Add fact to working memory
public void assertObject(Object obj) throws FactException;
// Modify copy of fact in working memory
public void modifyObject(Object obj) throws FactException;
// Remove fact from working memory
public void retractObject(Object obj) throws FactException;
```

Bemerkung. Wenn drools mein Projekt wäre, würde ich diese Methoden in `assertFact`, `modifyFact` und `retractFact` umbenennen.

Der Javacode in einem `consequence` hat Zugriff auf eine Instanz der Klasse `KnowledgeHelper` über eine Variable mit Namen `drools`.

6 Ein einfaches Beispiel

In einem leichten Beispiel simulieren wir ein Zoogeschäft mit den Geschäftsregeln unserer Einleitung Abschnitt 1.

Wir beginnen mit den Regeln, die im ersten Moment gar nicht so einfach aussehen, bis man gelernt hat, wie man sie lesen muß:

```
<?xml version="1.0"?>

<rule-set name="PetStore Rules"
  xmlns="http://drools.org/rules"
  xmlns:java="http://drools.org/semantics/java"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/rules rules.xsd
                    http://drools.org/semantics/java java.xsd">

  <import>javax.swing.JFrame</import>
  <import>java.math.BigDecimal</import>

  <application-data identifier="frame">JFrame</application-data>

  <java:functions>
    import org.drools.spi.KnowledgeHelper;
    import javax.swing.JOptionPane;
    import java.util.Iterator;
    import pets.*;

    <!-- assert all items in cart -->
    public void explodeCartAction(ShoppingCart cart, KnowledgeHelper drools)
    {
      System.out.println( "Examining each item in the shopping cart." );

      try
      {
        Iterator itemIter = cart.getItems().iterator();
        while (itemIter.hasNext())
        {
          drools.assertObject(itemIter.next());
        }
      }
    }
  }
</java:functions>
</rule-set>
```

```

        cart.setState("Exploded", true);
        drools.modifyObject(cart);
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}

<!-- put a free sample of fish food in cart -->
public void freeFishFoodSampleAction(ShoppingCart cart,
                                     KnowledgeHelper drools)
{
    try
    {
        System.out.println("Adding free Fish Food Sample to cart");
        cart.addItem(new pets.CartItem("Fish Food Sample",
                                       new BigDecimal("0.00")));

        drools.modifyObject(cart);
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}

<!-- ask customer if he would like to buy an aquarium -->
public void suggestTankAction(JFrame frame,
                              ShoppingCart cart,
                              KnowledgeHelper drools)
{
    Object[] options = {"Yes",
                       "No"};

    try
    {
        int n = JOptionPane.showOptionDialog(frame,
                                             "Would you like to buy a " +
                                             "tank for your " +
                                             cart.getItems("Gold Fish").size() +
                                             " fish?",
                                             "Purchase Suggestion",
                                             JOptionPane.YES_NO_OPTION,
                                             JOptionPane.QUESTION_MESSAGE,
                                             null,

```

```

                                options,
                                options[0]);
System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                + cart.getItems( "Gold Fish" ).size() +
                " fish? - " );
if (n == 0) {
    cart.addItem(new pets.CartItem("Fish Tank",
                                    new BigDecimal("25.00")));
    System.out.println( "Yes" );
} else {
    System.out.println( "No" );
}

    cart.setState("Suggested Fish Tank", true );
    drools.modifyObject(cart);
}
catch (Exception e)
{
    System.err.println(e);
}
}

<!-- give customer a 5% discount -->
public void apply5PercentDiscountAction(ShoppingCart cart,
                                        KnowledgeHelper drools)
{
    try
    {
        System.out.println("Applying 5% discount to cart");
        cart.setDiscount(new BigDecimal("0.05"));
        drools.modifyObject(cart);
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}

public void apply10PercentDiscountAction(ShoppingCart cart,
                                        KnowledgeHelper drools)
{
    try
    {
        System.out.println("Applying 10% discount to cart");
        cart.setDiscount(new BigDecimal("0.10"));
        drools.modifyObject(cart);
    }
}

```

```

    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}
</java:functions>

```

```

<!-- assert each item in the shopping cart into the Working Memory -->
<rule name="Explode Cart" salience="20">
    <parameter identifier="cart">
        <class>pets.ShoppingCart</class>
    </parameter>

    <java:condition>
        cart.getState("Exploded") == false
    </java:condition>

    <java:consequence>
        explodeCartAction(cart, drools);
    </java:consequence>
</rule>

```

```

<!-- Free Fish Food sample when we buy a Gold Fish if we haven't already
    bought Fish Food and dont already have a Fish Food Sample-->
<rule name="Free Fish Food Sample">
    <parameter identifier="cart">
        <class>pets.ShoppingCart</class>
    </parameter>
    <parameter identifier="item">
        <class>pets.CartItem</class>
    </parameter>

    <java:condition>
        cart.getItems("Fish Food Sample").size() == 0
    </java:condition>
    <java:condition>
        cart.getItems("Fish Food").size() == 0
    </java:condition>
    <java:condition>
        item.getName().equals("Gold Fish")
    </java:condition>

    <java:consequence>
        freeFishFoodSampleAction(cart, drools);
    </java:consequence>
</rule>

```

```

    </java:consequence>

</rule>

<!-- Suggest a tank if we have bought more than 5 gold fish and
dont already have one-->
<rule name="Suggest Tank" salience="10">
  <parameter identifier="cart">
    <class>pets.ShoppingCart</class>
  </parameter>

  <java:condition>
    cart.getState("Suggested Fish Tank") == false
  </java:condition>
  <java:condition>
    cart.getItems("Gold Fish").size() >= 5
  </java:condition>
  <java:condition>
    cart.getItems("Fish Tank").size() == 0
  </java:condition>

  <java:consequence>
    suggestTankAction(frame, cart, drools);
  </java:consequence>
</rule>

<!-- Give 5% discount if gross cost is more than 10.00
and less than 20.00 -->
<rule name="Apply 5% Discount">
  <parameter identifier="cart">
    <class>pets.ShoppingCart</class>
  </parameter>

  <java:condition>
    cart.getGrossCost().doubleValue() >= 10.00
  </java:condition>
  <java:condition>
    cart.getGrossCost().doubleValue() < 20.00
  </java:condition>
  <java:condition>
    cart.getDiscount().doubleValue() < 0.05
  </java:condition>

  <java:consequence>
    apply5PercentDiscountAction(cart, drools);

```

```

        </java:consequence>
</rule>

<!-- Give 10% discount if gross cost is more than 20.00 -->
<rule name="Apply 10% Discount">
  <parameter identifier="cart">
    <class>pets.ShoppingCart</class>
  </parameter>

  <java:condition>
    cart.getGrossCost().doubleValue() >= 20.00
  </java:condition>
  <java:condition>
    cart.getDiscount().doubleValue() < 0.10
  </java:condition>

  <java:consequence>
    apply10PercentDiscountAction(cart, drools);
  </java:consequence>
</rule>
</rule-set>

```

Kommentar:

1. Der aufmerksame Leser wird hier etliche Elemente bemerken, die bisher nicht erwähnt wurden:
 - mit einem `import`-Element kann man Klassen importieren, die in den Regeln benötigt werden.
 - ein `application-data`-Element bewirkt, daß der rule engine im working memory nachsieht, ob es dort eine Instanz der angegebenen Klasse gibt. Wenn ja, wird dieses Objekt dem Code in den `consequence`-Elementen unter dem im `identifizier`-Attribut angegebenen Namen verfügbar gemacht. Später werden wir sehen, wie unser Objekt `frame` in das working memory gelangt.
 - in einem `java:functions`-Element darf beliebiger Javacode stehen. In der Regel wird dieses Element dazu verwendet, Methoden zu definieren, die in den `consequence`-Elementen eingesetzt werden können. So halten wir es hier: jedes `consequence`-Element enthält den Aufruf einer derart definierten Methode. Dadurch werden unsere Regeln überschaubarer.
 - durch den Einsatz des `java:functions`-Elements werden unsere Regeln so leicht verständlich, daß man leicht kontrollieren kann, daß sie den in Abschnitt 1. formulierten Regeln entsprechen.
 - einige von unseren Regeln haben hier ein Attribut `salience` mit einem ganzzahligen Wert. Das Wort “salience” könnte man mit “Wichtigkeit” oder “Relevanz” übersetzen. Die Semantik an dieser Stelle besteht darin, der Regel eine Prioritätsstufe zuzuweisen. Die Regel mit Namen `Explode Cart` hat einen

`salience`-Wert von 20 und somit die höchste Priorität aller Regeln, denn wir wollen, daß sich der Inhalt des Warenkorb im working memory befindet, bevor `drools` versucht, andere Regeln zu testen.

7 Die Javaklassen

Um den Javacode in unserem Regeldokument zu verstehen, muß man die betreffenden Javaklassen kennen. Es folgen die wichtigsten (d. h. mit Ausnahme der Klasse `PetstoreUI`, die die Benutzeroberfläche konstruiert). Wir beginnen mit der Klasse `CartItem`:

```
package pets;

import java.math.BigDecimal;

public class CartItem
{
    private String name;

    private BigDecimal cost;

    public CartItem(String name, BigDecimal cost)
    {
        this.name = name;
        this.cost = cost;
    }

    public String getName()
    {
        return this.name;
    }

    public BigDecimal getCost()
    {
        return this.cost;
    }

    public String toString()
    {
        return name + " " + this.cost;
    }
}
```

Und nun die Klasse `ShoppingCart`:

```
package pets;
```

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.math.BigDecimal;
import java.math.RoundingMode;

public class ShoppingCart
{
    private List<CartItem>        items;
    private BigDecimal            discount;

    /**
     * This is just a convenient storage area, in which the rule
     * engine can remember what it has/has not done so far.
     */
    private Map<String, Boolean> states;

    private static String newline = System.getProperty("line.separator");

    public ShoppingCart()
    {
        states = new HashMap<String, Boolean>( );
        this.items = new ArrayList<CartItem>( );
        this.discount = new BigDecimal("0.00");
    }

    public boolean getState(String state)
    {
        if (states.containsKey(state))
        {
            return states.get(state);
        }
        else
        {
            return false;
        }
    }

    public void setState(String state, boolean value)
    {
        states.put(state, value);
    }
}

```

```

public void setDiscount(BigDecimal discount)
{
    this.discount = discount;
}

public BigDecimal getDiscount()
{
    return this.discount;
}

public void addItem(CartItem item)
{
    this.items.add(item);
}

public List<CartItem> getItems()
{
    return this.items;
}

public List<CartItem> getItems(String name)
{
    ArrayList<CartItem> matching = new ArrayList<CartItem>( );

    for (CartItem item : items)
    {
        if (item.getName().equals(name))
        {
            matching.add(item);
        }
    }

    return matching;
}

public BigDecimal getGrossCost()
{
    BigDecimal cost = new BigDecimal("0.00");

    for (CartItem item : items)
    {
        cost = cost.add(item.getCost());
    }

    return cost;
}

```

```

    }

    public BigDecimal getDiscountedCost()
    {
        BigDecimal cost = getGrossCost( );
        BigDecimal discount = getDiscount( );
        BigDecimal one = new BigDecimal("1.00");
        BigDecimal factor = one.subtract(discount);

        BigDecimal discountedCost = cost.multiply(factor);

        return discountedCost.setScale(2,RoundingMode.HALF_UP);
    }

    public String toString()
    {
        StringBuffer buf = new StringBuffer( );

        buf.append("ShoppingCart:" + newline);

        Iterator itemIter = getItems( ).iterator( );

        while ( itemIter.hasNext( ) )
        {
            buf.append( "\t" + itemIter.next( ) + newline );
        }

        buf.append("gross total=" + getGrossCost( ) + newline);
        buf.append("discounted total=" + getDiscountedCost( ) + newline);

        return buf.toString( );
    }
}

```

Kommentar: Der Code dieser Klasse dürfte selbsterklärend sein – mit möglicher Ausnahme des Attributs `states`; aber dieses ist wiederum mit einem Kommentar versehen, der aufklären dürfte, was die Rolle dieses sonst so rätselhaften Attributs sein soll. Wenn der Leser zu unserem Regeldokument zurückblättert, wird er Beispiele finden, wo dieses Attribut `states` eingesetzt wird.

Als nächste sehen wir die Klasse `PetStore`, die den rule engine in Aktion setzt:

```

package pets;

import java.util.Vector;

```

```

import org.drools.RuleBase;
import org.drools.io.RuleBaseLoader;

import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.math.BigDecimal;

public class PetStore
{
    public static void main(String[] args)
    {
        if ( args.length != 1 )
        {
            System.out.println("Usage: " + PetStore.class.getName( )
                               + " [drl file]");
            return;
        }
        System.out.println("Using drl: " + args[0]);

        try
        {
            FileInputStream fin = new FileInputStream(args[0]);
            InputStreamReader reader = new InputStreamReader(fin);
            RuleBase ruleBase = RuleBaseLoader.loadFromReader(reader);
            if (ruleBase == null)
            {
                System.err.println("could not load ruleBase");
                System.exit(-1);
            }

            Vector<CartItem> stock = new Vector<CartItem>( );
            stock.add( new CartItem("Gold Fish", new BigDecimal("5.00")));
            stock.add( new CartItem("Fish Tank", new BigDecimal("25.00")));
            stock.add( new CartItem("Fish Food", new BigDecimal("2.00")));

            //The callback is responsible for populating working memory and
            // fireing all rules
            PetStoreUI ui = new PetStoreUI(stock,
                                           new CheckoutCallback(ruleBase));
            ui.createAndShowGUI( );
        }
        catch (Exception e)
        {
            System.err.println("Error in PetStore.main");
            System.err.println(e);
        }
    }
}

```

```

    }
}
}

```

Kommentar: Die Art, wie hier auf den rule engine zugegriffen wird, ist absolut drools-spezifisch. In [jsr94] findet man keine Klasse RuleBase.

Und zu guter Letzt sehen wir die Callbackklasse CheckoutCallback, die in Aktion tritt, wenn der Kunde “zur Kasse geht” – d. h. auf den Button mit der Aufschrift Checkout klickt:

```

package pets;

import java.util.List;

import javax.swing.JFrame;

import org.drools.FactException;
import org.drools.RuleBase;
import org.drools.WorkingMemory;

/**
 *
 * This callback is called when the user presses the checkout button. It is
 * responsible for adding the items to the shopping cart, asserting the shopping
 * cart and then firing all rules.
 *
 * A reference to the JFrame is also passed so the rules can launch dialog boxes
 * for user interaction. It uses the ApplicationData feature for this.
 *
 */
public class CheckoutCallback
{
    RuleBase ruleBase;

    public CheckoutCallback(RuleBase ruleBase)
    {
        this.ruleBase = ruleBase;
    }

    /**
     * Populate the cart and assert into working memory Pass JFrame reference
     * for user interaction
     *
     * @param frame
     * @param items
     */
}

```

```

    * @return cart.toString();
    */
public String checkout(JFrame frame, List<CartItem> items)
    throws FactException
{
    ShoppingCart cart = new ShoppingCart( );

    //Iterate through list and add to cart
    for (CartItem item : items)
    {
        cart.addItem(item);
    }

    //add the JFrame to the ApplicationData to allow for user interaction
    WorkingMemory workingMemory = ruleBase.newWorkingMemory( );
    workingMemory.setApplicationData("frame", frame);
    workingMemory.assertObject(cart);
    workingMemory.fireAllRules( );

    //returns the state of the cart
    return cart.toString( );
}
}

```

Kommentar:

1. Wir sehen hier die Instruktion

```
workingMemory.setApplicationData("frame", frame);
```

wo unser Anwendungsobjekt vom Typ JFrame und mit Namen `frame` ins working memory eingefügt wird. In unserem Regeldokument haben wir gesehen, wo wir dieses Objekt abrufen.

2. Danach folgt die Instruktion

```
workingMemory.assertObject(cart);
```

mit der wir unseren virtuellen Einkaufswagen ins working memory schreiben. Nur dort können die Regeln auf ihn wirken.

3. Und zuletzt folgt die Instruktion

```
workingMemory.fireAllRules( );
```

die den rule engine veranlaßt, alle Regeln anzuwenden, deren Bedingungen erfüllt werden. Anfangs sind das nicht viele (nämlich nur die Regel mit Namen

Explode Cart

4. Der Leser wird sich evtl. fragen, wieso die Regel mit Namen `Free Fish Food Sample` anfangs nicht anwendbar ist. Die Antwort liegt in der Art, wie `drools` mit dem `parameter`-Element umgeht. Dieses Element bewirkt, daß der rule engine im working memory nachsieht, ob es eine Instanz der genannten Klasse gibt (in unserem Fall vom Typ `CartItem`). Ist das nicht der Fall gelten die Bedingungen der betreffenden Regel von vornherein als *nicht* erfüllt. Dies macht es übrigens überflüssig, eine `java:condition` folgender Art vorzusehen:

```
<java:condition>item != null</java:condition>
```

Ein solches Element vom Typ `CartItem` befindet sich erst nach Ausführung der Regel `Explode Cart` im working memory.

8 Ein zweites Beispiel

Jetzt wollen wir uns ein zweites Beispiel für die Anwendung eines rule engine anschauen. Auf den ersten Blick scheint dieses zweite Beispiel einfacher zu sein als das erste – aber nur auf den ersten Blick, denn hier gibt es einen subtilen Punkt, der nur durch Anwendung der in `drools` eingebauten *Konfliktauflösung* in den Griff zu bekommen ist. Hier brauchen wir unbedingt die durch das Attribut `salience` spezifizierte Präzedenz.

In diesem Beispiel wollen wir mit dem rule engine die jedem Informatiker vertraute Fibonacci-Funktion $fib(n)$ berechnen. Jeder kennt die Regeln, die wir so formulieren können:

1. Anfang der Rekursion:

(a) $fib(0) = 0$

(b) $fib(1) = 1$

2. Rekursionsschritt: Für jedes $n \geq 2$ gilt

$$fib(n) = fib(n - 1) + fib(n - 2)$$

In der Sprache DRL lauten unsere Regeln so:

```
<?xml version="1.0"?>
```

```
<rule-set name="fibonacci"
  xmlns="http://drools.org/rules"
  xmlns:java="http://drools.org/semantics/java"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/rules rules.xsd
```

<http://drools.org/semantics/java/java.xsd>>

```
<import>fibonacci.Fibonacci</import>

<rule name="Bootstrap 1" salience="20">
  <parameter identifier="f">
    <class>Fibonacci</class>
  </parameter>

  <java:condition>f.getIndex() == 1</java:condition>
  <java:condition>f.getValue() == -1</java:condition>
  <java:consequence>
    f.setValue(1);
    System.err.println("fib(" + f.getIndex() + ") == " + f.getValue());
    drools.modifyObject(f);
  </java:consequence>
</rule>

<rule name="Bootstrap 2">
  <parameter identifier="f">
    <class>Fibonacci</class>
  </parameter>
  <java:condition>f.getIndex() == 2</java:condition>
  <java:condition>f.getValue() == -1</java:condition>
  <java:consequence>
    f.setValue(1);
    System.err.println("fib(" + f.getIndex() + ") == " + f.getValue());
    drools.modifyObject(f);
  </java:consequence>
</rule>

<rule name="Recurse" salience="10">
  <parameter identifier="f">
    <class>Fibonacci</class>
  </parameter>
  <java:condition>f.getValue() == -1</java:condition>
  <java:consequence>
    System.err.println("recurse for " + f.getIndex());
    drools.assertObject(new Fibonacci( f.getIndex() - 1 ));
  </java:consequence>
</rule>

<rule name="Calculate">
  <parameter identifier="f1">
    <class>Fibonacci</class>
```

```

</parameter>
<parameter identifier="f2">
  <class>Fibonacci</class>
</parameter>
<parameter identifier="f3">
  <class>Fibonacci</class>
</parameter>
<java:condition>f2.getIndex() == (f1.getIndex() + 1)</java:condition>
<java:condition>f3.getIndex() == (f2.getIndex() + 1)</java:condition>
<java:condition>f1.getValue() != -1</java:condition>
<java:condition>f2.getValue() != -1</java:condition>
<java:condition>f3.getValue() == -1</java:condition>
<java:consequence>
  f3.setValue(f1.getValue() + f2.getValue());
  System.err.println("fib(" + f3.getIndex() + ") == " + f3.getValue());
  drools.modifyObject(f3);
  drools.retractObject(f1);
</java:consequence>
</rule>

```

```
</rule-set>
```

Um diese Regeln wirklich zu verstehen, müssen wir unsere Klasse Fibonacci kennen:

```

package fibonacci;

import java.io.Serializable;

public class Fibonacci implements Serializable
{
    private static final long serialVersionUID = -4073246884111449193L;
    private int index;

    private long value;

    public Fibonacci(int index)
    {
        this.index = index;
        this.value = -1;
    }

    public int getIndex()
    {
        return this.index;
    }
}

```

```

public void setValue(long value)
{
    this.value = value;
}

public long getValue()
{
    return this.value;
}

public String toString()
{
    return "Fibonacci(" + this.index + "/" + this.value + ")";
}
}

```

Fibonacci ist ein POJO, das einen Wert der Fibonacci-Funktion repräsentiert. Es hat zwei Attribute

- `index` ist das n in $fib(n)$.
- `value` ist der Wert von $fib(n)$.

Objekte dieses Typs werden im working memory gespeichert; zunächst hat `value` den Wert -1 . Wenn der rule engine ein solches Objekt vorfindet berechnet er den richtigen Wert aufgrund der Werte anderer solcher Objekte und setzt den neuen Wert ein.

Und dies ist der Punkt, wo wir unsere Regeln mit Prioritäten versehen müssen; wir wollen sichergehen, daß der Rekursionsanfang erledigt wird, bevor der Rekursionsschritt in Angriff genommen wird. Wenn der Leser zu unserem Regeldokument zurückblättert, wird er feststellen, daß die Regel `Bootstrap 1` eine höhere Priorität bekommt (20) als die Regel `Recurse (10)`.

Die Klasse, die unsere Anwendung in Gang setzt heißt `FibonacciExample` und sieht so aus:

```

package fibonacci;

import java.io.*;
import org.drools.RuleBase;
import org.drools.WorkingMemory;
import org.drools.io.RuleBaseLoader;

public class FibonacciExample
{
    public static void main(String[] args)
    {
        if (args.length != 1)

```

```

{
    System.out.println("Usage: " + FibonacciExample.class.getName( )
        + " [drl file]");
    return;
}
System.out.println("Using drl: " + args[0]);

try
{
    FileInputStream fin = new FileInputStream(args[0]);
    InputStreamReader reader = new InputStreamReader(fin);
    RuleBase ruleBase = RuleBaseLoader.loadFromReader(reader);
    if (ruleBase == null)
    {
        System.err.println("could not load ruleBase");
        System.exit(-1);
    }

    WorkingMemory workingMemory;
    Fibonacci fibonacci;
    long start;
    long stop;

    System.err.println("\nFirst run - code compiled on the fly");
    workingMemory = ruleBase.newWorkingMemory( );

    fibonacci = new Fibonacci(50);
    start = System.currentTimeMillis( );
    workingMemory.assertObject(fibonacci);

    workingMemory.fireAllRules( );
    stop = System.currentTimeMillis( );

    System.err.println("fibonacci(" + fibonacci.getIndex( )
        + ") == " + fibonacci.getValue( ) + " took "
        + (stop - start) + "ms");
    System.err.println("\nSecond run - code already compiled");

    workingMemory = ruleBase.newWorkingMemory( );

    fibonacci = new Fibonacci(50);
    start = System.currentTimeMillis( );
    workingMemory.assertObject(fibonacci);

    workingMemory.fireAllRules( );

```

```

        stop = System.currentTimeMillis( );
        System.err.println("fibonacci(" + fibonacci.getIndex( )
            + ") == " + fibonacci.getValue( ) + " took "
            + (stop - start) + "ms");
    }
    catch (Exception e)
    {
        e.printStackTrace( );
    }
}
}

```

Kommentar:

1. Wir sehen hier das bereits vertraute Schema, mit dem man unter `drools` den Zugriff auf einen lokalen rule engine (ohne JNDI) bekommt.
2. Danach fügen wir ein Objekt vom Typ `Fibonacci` mit `index 50` und `value -1` in das working memory ein. Der Aufruf `fireAllRules` veranlaßt den rule engine die Regel `Recurse` anzuwenden, da zu diesem Zeitpunkt diese die einzige anwendbare Regel ist.
3. Diese Regel arbeitet sich rückwärts durch und erzeugt Objekte mit `index` zwischen 49 und 1 und fügt diese ebenfalls ins working memory ein.
4. Nun kann die Regel `Bootstrap 1` angewendet werden; da sie eine höhere Priorität besitzt als `Recurse`, wird `Recurse` nicht angewendet, um ein Objekt mit `index 0` zu erzeugen.
5. Nun ist `Bootstrap 2` anwendbar und berechnet den korrekten Wert für das Objekt mit `index 2`.
6. Schließlich und endlich kommt die Regel `Calculate` zum Zuge und berechnet die richtigen Werte für die Objekte mit `index 3...50`. Nun kann der rule engine ruhen.

Literatur

- [jsr94] <http://www.jcp.org/aboutJava/communityprocess/review/jsr94/>. Hier findet man den JSR 94.